Code Coverage Analysis of Object-Oriented Programming

A thesis submitted in partial fulfilment of the requirements for the degree of

Bachelor of Technology

in

Computer Science and Engineering

by

Hemant Singh

(Roll no. 107cs003)

and

Vikram Singh Bhati

(Roll no. 107cs013)

Under the guidance of :

Prof. D.P. Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769 008, Orissa, India

National Institute of Technology
Rourkela

Certificate

This is to certify that the project entitled, '**Code Coverage Analysis of Object-Oriented Programming**' submitted by **Hemant Singh** and **Vikram Singh Bhati** is an authentic work carried out by them under my supervision and guidance for the partial fulfillment of the requirements for the award of **Bachelor of Technology Degree** in **Computer Science and Engineering** at **National Institute of Technology, Rourkela**.

To the best of my knowledge, the matter embodied in the project has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date - 6/5/2011
Rourkela

(Prof. D.P. Mohapatra)
Dept. of Computer Science and Engineering

Acknowledgement

On the submission of our Thesis report, we would like to extend our gratitude and sincere thanks to our supervisor Prof. D.P. Mohapatra, for his constant motivation and support during the course of our work in the last one year. He has been the corner stone of our project and has guided us during periods of doubts and uncertainties. We truly appreciate and value his esteemed guidance and encouragement from the beginning to the end of this thesis. He has been our source of inspiration throughout the thesis work and without his invaluable advice and assistance it would not have been possible for us to complete this thesis.

A special acknowledgement goes to Mrs. Mitrabinda Ray, a PhD scholar for her guidance throughout the thesis. Finally, we also thank all the professors of our department for being a constant source of inspiration and motivation during the course of the project.

Date - 6/5/2011
Rourkela

Hemant Singh

Vikram Singh Bhati

Abstract

Code coverage analysis is a vital activity in any software testing process. It provides developers with a measure of how well their source code is being exercised by the test runs.It estimates the effectiveness of the test by detecting errors/bugs in the code. In order to perform efficient software testing and coverage analysis we must adopt a systematic way and a sound theoretical basis for testing the programs. In our project, we are describing the implementation of a testing tool, Java Bytecode Understanding and Testing(JaBUTi) tool for testing Java programs. JaBUTi works with Java Bytecode so no source code is required for the code coverage analysis. It shows the Definition-Use Graph(DUG) for every method of a class which helps in the structural testing approach, both control flow testing and data flow testing. It consists of a code coverage tool, slicing tool and static metrics tool. In our project, we are using JaBUTi as a code coverage tool for coverage analysis of Java programs. We have proposed an algorithm for coverage analysis of code to desired percentage. In our algorithm we are taking Java Bytecode as an input. With the help of Graphviz, another tool for the graphical representation of programs, JaBUTi generates the DUG for every constituent methods of a class. The DUG representation highlights the various nodes with their priority information. JaBUTi executes a Test Case file, which consists of test cases, to measure the percentage of the code covered. The Test Case file is continuously updated after each run by adding a test case to it. A test run is considered when it increases the coverage in the consecutive run otherwise the Test Case file is updated with a new test case. The process continues until the code coverage exceeds the required coverage percentage. We have described the working of this algorithm through a number of case studies. Our analysis is mainly based on the All-Nodes-exception independent(All-Nodes-ei) criteria. At the end, we have also made a comparison between the coverage and the test runs for each case study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the continuous improvement in the Component-Based Software Development[5], the component user always needs to check the quality of the software components being integrated. Software components inherit most of the characteristics of the Object-Oriented programming, but the notion of components exceeds the notion of objects. In object oriented programming, components can be defined as a block of code, a class etc.

Code coverage analysis is a quantitative approach to measure what amount of the source code is being covered. Code coverage is most often used during the software testing process in order to determine the coverage achieved by the test cases and therefore it is also referred to as test coverage. Code coverage analysis is performed by first instrumenting the source code or intermediate binaries to output coverage information during its execution.

In our simulation we have used the control-flow and data-flow criteria to support the testing of Java programs (Java bytecode) aimed at intra-method structural testing of code and its components based on various testing criteria. A testing tool, named JaBUTi (Java Bytecode Understanding and Testing), which supports the application of such criteria for testing Java programs and components, is used for our structural testing and code coverage analysis.

1.1 Motivation for our project

Testing process is carried out in various levels to uncover all the existing defects in the software product. However, most of the times, even after satisfactorily carrying out the testing process, we cannot ensure that a software product is error free. This situation is caused by the fact that the domain of the input data for most of the software products is very large. Hence, it is practically impossible to test the software product exhaustively with all the test cases. It is quite obvious that not all the lines in the source code contribute to the error at a particular location. We are therefore not required to contribute the whole source code in the testing process and our only focus are on those areas that are more likely to have caused the error. In order to find these high-risk areas, we need to construct an intermediate representation of a sample input program for eg. Definition-Use graph and analyse the code coverage by distributing the testing efforts accordingly.

1.2 Objective of our project

Our objective is to propose an algorithm to estimate the high code coverage of Java programs by writing the minimum number of suitable test cases. In this project we will implement our proposed algorithm to achieve the overall code coverage of more than 90 percent for each case study. In order to increase the code coverage with the minimum test runs, we will use the DUG(Definition-Use graph) representation of every methods in a Java program. We will first exercise the high priority nodes of DUG and then moving to the lower priority nodes. We will also compare the estimated coverage with the number of test runs for each case study. Finally an overall assessment of all the case studies will be done.

1.3 Organization of the project report

This report has been organized into seven chapters.

Chapter 2

In this chapter, we discuss the related work done in the field of code coverage analysis.

Chapter 3

In this chapter, we present a basic overview of code coverage analysis. The definition, need and the advantages of the code coverage analysis is described in this chapter. Various criteria and metrices like branch coverage, path coverage, statement coverage etc,which are followed in code coverage determination are also elaborated.

Chapter 4

In this chapter,we present the basic concepts related to the intermediate representation of a program. Three different ways of graphical representation of any input program namely, Control flow graph, Data flow graph and Definition Use Graph, are described in this chapter.

Chapter 5

In this chapter,we propose an algorithm which describes step by step procedure for calculating the code coverage of a Java program is described.

Chapter 6

In this chapter,we describes the various tools required in the project like Jabuti, Netbeans 6.9 and Graphviz. Also the simulation of various examples are shown here with screen shots. Four examples namely, Vending Machine Example, Subject Record Example, Factor calculator Example and Banking System Example are explained here with the simulation results. The Coverage graph of all examples with respect to the test runs are also plotted.

Chapter 7

In this chapter, we conclude the project and discuss the future work that can be done in this area.

Chapter 2

Related Work

In 1999, Bor-Yuan Tsai, Simon Stobart, Norman Parrington, and Ian Mitchell[14] proposed a novel Object-Oriented class testing approach which is a combination of functional and structural testing techniques. A state-based testing is implemented and the test cases generated from the MACT (Method for Automatic Class Testing) tool can be used to execute functional testing. In this new class testing approach for Object-Oriented classes, the test case tree, created from the state machine of the class under test, produces test messages for functional testing as well as for the intra-class definition-use information for structural testing.

In 2000, Jianjun Zhao[10] presented a paper on dependence analysis of Java bytecode,this paper emphasizes on understanding the program dependencies in a program by analysing control flow and data flow which further helps in the debugging and testing process. It also analyses both control and data dependencies in a bytecode program and highlights the need to test java program at bytecode level.

In December 2002, Yashwant, Michael and Rick[15] presented a LE (Log-arithmetic-exponential) model which emphasizes on testing efforts to test coverage (which includes block coverage, branch coverage, computation-use, or predicate-use). This model is based on the time based logarithmic software growth model. Their model relates test-coverage measure directly with the defect-coverage and considers that 100% coverage might not ensure uncovering of all the defects.

Most softwares are tested in one of two ways: (1) functional (black-box) testing, and (2) structural (white-box) testing. The demerit of functional techniques is that it may not provide sufficient code coverage, whereas structural approaches do not consider the requirements of specifications at all, since all the test cases are generated from the implementation [1].

In 2004, Vincenzi, Maldonado, Wong and Delmaro[3] presented a paper on coverage testing of Java programs and components. In this paper they have described the control flow and data flow based criteria for coverage testing of java components. Control flow testing is described using all-nodes and all-edges criteria with the help of the DUG representation of program. Data flow testing is described based on the all-uses criteria comprising of all-c-uses and all-p-uses criteria. They have used Jabuti as a testing tool and have shown its implementation for control flow and data flow testing.

Here we are using JaBUTi as a primary tool for the Code coverage estimation of Object-Oriented programming. Test cases are built in another tool called Netbeans 6.9. For the DUG visualization we are taking help of another tool called Graphviz. We have proposed an algorithm to analyse the coverage of programs in JaBUTi. Different cases studies have been taken and their respective code coverage is sumarized by different criteria like by method, by class etc. We have done the implementation for All-Nodes-ei criteria. Test case file written in Netbeans IDE is imported in JaBUTi and test cases are constantly added to it using Netbeans in order to increase the coverage until required coverage is obtained. The work and implementation is shown in the following chapters.

Chapter 3

Coverage Analysis

3.1 Overview

Code coverage is a quantitative measure which determines the extent of the source code of a program that has been exercised or covered. It is most frequently used during the software testing process in order to determine the coverage achieved by the testing process. The tools used to perform the code coverage first instrument the source code or intermediate binaries with instructions and then outputs the coverage information during its execution. The coverage tool then gathers the generated data or adds the information to some log file and reference the original source code to produce a coverage report.

Coverage reports mainly contains information about the original source code indicating how frequently each element has been covered. It also indicates overall coverage level each mentioned under different criteria like by class, by methods etc. The coverage levels are defined for individual classes, modules and functions.
The coverage score gives a general view of how much code is covered by the Test cases. A low coverage score indicates an inadequate level of testing which implies a lower probability of the test cases to uncover any bugs in the code. In addition, a low initial coverage level also indicates a deficiency in the test development process. Also while a good test suite is likely to score high coverage levels, even tests with the highest possible score of coverage are not guaranteed to catch all errors[15]. It is natural to treat coverage levels as a direct measure of judging the test effectiveness but

it is important to instead think of it as a tool to better understand the conduct and shortcomings of the testing process and the tests at hand. Furthermore, by including coverage analysis as a process within the software development cycle, developers can avoid the circumstances of the code evolving well beyond what is being interrogated by the tests, resulting in an increasingly obsolete set of tests that may lead to a misplaced sense of confidence in the code.

3.2 Criteria and Metrics

The data collected from coverage analysis can be used to measure and improve the test process. Coverage analysis can be performed in a variety of ways for producing different views of how a test process executes the code. To go for coverage analysis, one or more of the following criteria are commonly used.

Function coverage analysis: It involves invoking each and every function or procedure of the code at least once.

Statement coverage analysis: It involves invoking of each and every executable statement (which can be a multiple statements in a single line) run at least once. Statement coverage analysis is the basic form of coverage analysis.

Branch coverage analysis: It involves the coverage of every control structure by evaluating them to both true and false at least once. Branch coverage analyses each branch to determine the possible paths through it and produces data to show which paths were taken and how often[2]. For example, in the code below

```
If(t == true)
{
Do something
}
else
```

{

Do something else

}

Relational operator coverage analysis: The relational operators (>, ≤, ≥,<) are often the Cause of faults, whether because of swapped operators (e.g., ≤ instead of ≥), incorrect use (e.g., <instead of≤), or incorrect boundary values (e.g., a <9 instead of a <10).

Condition coverage analysis: It involves every logical condition within a control structure evaluated to both true and false at least once.

Path coverage analysis: It involves coverage of every possible route through the program taken at least once. A path is a unique sequence of logical conditions which means that full path coverage analysis requires attempting every combination of logical conditions within the program. For example, in code below

if (a)

{

Do something

}

if (b)

{

Do something else

}

In this a full path coverage analysis is done by setting the value of (a,b) as (0,0), (0,1), (1,1), (1,0).

Chapter 4

Basic Concepts

In this chapter we discuss the basic concepts and terminologies related to our work. Three types of graph CFG , DFG , DUG that can be used for the intermediate representation of a program are explained here:

4.1 Control Flow Graph

A Control Flow Graph can be defined as a directed graph with a unique entry node START and a unique exit node STOP, where each node is a statement in the program. There is a directed edge from node A to node B in the CFG if control may flow from block A directly to block B. Edges in a CFG are of two types, reguler and exception. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

```
a = 5;
cnt = 20;
while(cnt > 0)
{
        if(a < 15)
        {
                a=a+1;
        }
        cnt = cnt – 1;
}
```

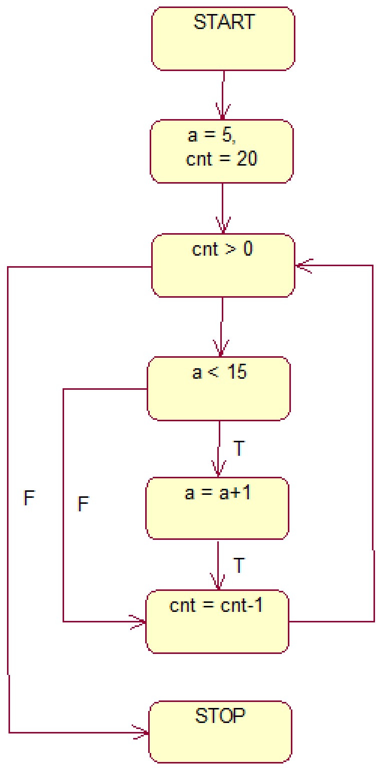


Figure 4.1: Control flow graph for the given piece of code

4.2 Data Flow Graph

A data-flow graph or DFG is a graphical representation of data dependencies between a number of functions

If node Y is data dependent on node X, then X is called the reaching definition of Y.

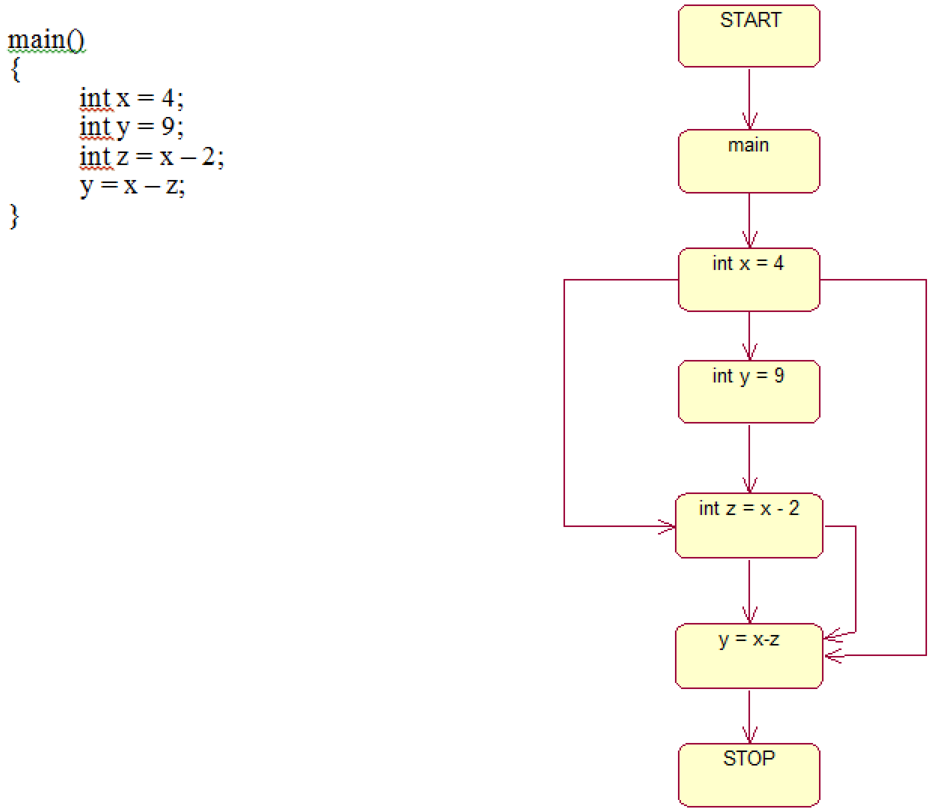


Figure 4.2: Data Flow graph for the given piece of code

4.3 Definition-Use Graph

Data-flow testing criteria use the def-use graph (DUG), which is an extension of the CFG. It contains the information about the set of variables defined and used in each node/edge of the CFG[6]. The Def-Use Graph is the underling model used in Jabuti to derive both control-flow and data-flow testing requirements. On the DUG we have two different types of edges:

1. Regular edges consisting of continuous lines that represent the regular control-flow.
2. Exception edges consisting of dashed lines representing the control-flow when an exception is raised.

Also,in the DUG graph, termination nodes are represented by bold line circles and regular nodes are represented by single line circles.

Chapter 5

Proposed algorithms

For a efficient code coverage algorithm, it must fulfils both the following criteria.

1. High code coverage
2. Minimum test runs

A high code coverage with a large number of test run is also an inefficient way for overage analysis. min test runs and low coverage is also inefficient in uncovering errors/bugs in the code. So we need a step by step approach for covering more code with minimum test runs. Here we have proposed an algorithm which enhances the code coverage in minimum test runs by covering the higher priority nodes first and then proceeding to lower priority nodes.

In this algorithm we are using a test case file for adding the test cases. When this file is executed, all the test cases in the file are executed. We have an input java program whose coverage we have to measure. When this class file is given as an input to JaBUTi, we can generate the DUG implementation of every method Now initially, some test cases are added to the test file and it is run to measure the initial coverage(for eg. one test case is added). Now with the help of DUG implementation we can find out the various lines of code(or DUG node) with different priority. Test case corresponding to higher priority node is added to the Test Case file and the file is executed. Corresponding coverage is determined. If the the coverage after the test run is same as before, test case is rejected without incrementing the number of test runs. If the coverage increases, then the test case is accepted and the number of test run is incremented by 1. We proceed step by step in the same way until the coverage increases the required level. When coverage increases the required level, we get the minimum number of test runs and the corresponding coverage.

Improved Code Coverage We are using following variables for representing different terms.

D : Test cases domain containing n test cases

f_test : test case file which is executed

N : total number of test runs

Cov : total coverage of the program

Req_Cov: required coverage value of program

Algorithm in the form of pseudo code is shown below:

1. *Generate the DUG of the input java program using JaBUTi.*
2. *Initialize array total_coverage[] to 0 and variable Cov to 0.*
3. *Initialize N to 0.*
4. *Write initial test cases to the f_test and run it to measure the initial coverage.*
5. *Add the test case from the D to the f_test based on high priority DUG nodes*
6. *Run the f_test*
7. *If (total_coverage[i] >total_coverage[i-1])*

```
{
        N = N + 1 and
        Cov = total_coverage[i].
if (Cov >Req_Cov )
{
        return Cov and N.
}
else
{
        continue from Step(5).
}
}
else
{
        Do nothing and continue from Step(5).
```

}

8. *Return Cov and N.*

Explanation: This algorithm is a simple way to proceed for code coverage analysis of any program. Initially some test cases are written in file f_test and then it is run to measure the coverage, Cov. If the value of Cov after a run is same as before the run, which means that despite of adding a test case the coverage has not increased, then we do not increment the number of test runs, N. If the consecutive run values of Cov increases then the value of N is incremented by 1. Once the value of Cov exceeds the Req_Cov value, the process stops and the values of Cov and N are returned.This algorithm can be implemented for the code coverage of Object-Oriented Program for any of the six-criteria that JaBUTi defines namely, All-Nodes-ei, All-edges-ei,All-uses-ei, All-Pot-Uses-ei, All-Nodes-ed, All-edges-ed, All-uses-ed, All-Pot-Uses-ed.In our project we have implemented the algorithm for All-Nodes-ei criteria. Same is applicable for other criteria.

Chapter 6

Simulation and Result

6.1 Toolsuite Used

6.1.1 Java Bytecode Understanding and Testing (JaBUTi)

JaBUTi is a set of tools designed for understanding and testing of Java programs. JaBUTi consists of

1. Code coverage analysis testing tool
2. Slicing tool
3. Static metrics tool

Here JaBUTi is used only as a code coverage analysis testing tool. The coverage tool can be used to assess the quality of a given test set or to generate test set based on different control-flow and data-flow testing criteria[3].

JaBUTi tool consists of a test driver that instruments the classes loaded for coverage analysis and executes it. The test driver consists of a class loader that instruments the original classes by calling a static method probe() of DefaultProber class in probe package that stores the information about the piece of code about to be executed. At the end of the program, another method dump() of DefaultProber class in probe package is called and all the data collected by the calls to probe() function are written to a trace file.

JaBUTi provides the visualization of the testing requirements at the bytecode, source code and DUG graph of each method, for all the supported criteria. Colors are associated with the testing requirements determining their weights.

Advantages

1. It uses the Java byte code (.class files) instead of the Java source code to collect the information necessary to perform its activities. This characteristic allows the tool to be used to test any kind of Java program, including Java components.
2. It enables the structural testing of Java components and also the identification of which part of such components needs more test or has not yet been covered.
3. Even when no source code is available, the tester can at least identify which part of the byte code requires more test, or, in case of a fault is identified, which part of the component is more probable to contain that fault.
4. It uses different colors to provide hints to the tester to ease the test generation.

6.1.2 Netbeans 6.9

The NetBeans IDE is written in Java comprising an integrated development environment and an extensible plug-in system and can run anywhere a JVM is installed, including Windows, Mac OS, Linux, and Solaris. A JDK is required for Java development functionality, but is not required for development in other programming languages. The NetBeans platform allows applications to be developed from a set of modular software components called modules. It is easy to create java application with the help of Netbeans. We have used Netbeans here to write the Junit Test cases for the code coverage analysis. The test cases written in Netbeans is imported to Jabuti where its code coverage is assessed for all the instrumenting classes.

6.1.3 Graphviz

Graphviz is a tool used for the pictorial representation of a graph. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. We have used this tool in our project to visualize DUG for every function whose coverage is to be determined with the help of test cases. This tool converts the structural representation of a function to the graph consisting of nodes and edges.

Nodes determine one or more line of code and edges determines the control transfer between the instructions. The Graphviz reads from the output file in order to generate the graph that the user can visualize [5].

6.2 Case Studies

6.2.1 The FactorCalculator example

This example comprises of three methods isPrime(), factor() and isDivisor(). The isDivisor() method takes two numbers as arguments and checks whether one divides other or not. The isPrime() method takes one argument and checks whether the number is prime or not. The factor() method takes one argument and calculates its factors. The factor() and isPrime method also calls the method isDivisor() to check the divisibility of two numbers.
The Test case FactorTestCase.java is implemented for writing test cases required for the desirable code coverage. The test file is included in the JaBUTi and processed. After every run we check for the code covered by the test case and to increase the code coverage, the different test cases are added to it and constantly run until the required code coverage is obtained. The analysis tool constantly polls the trace file. When a new test case is appended in the trace file, the tool updates the test information, recomputing the coverage. Jabuti also provides the several features to perform code visualization, DUG graph visualization and report generation . A program/component test is organized in a test project, indicating the set of classes under test that can be saved and re-loaded by the tester.

Following are the various results obtained for FactorCalculator example
Total number of Classes = 1
Total lines of source code = 80

Screenshots

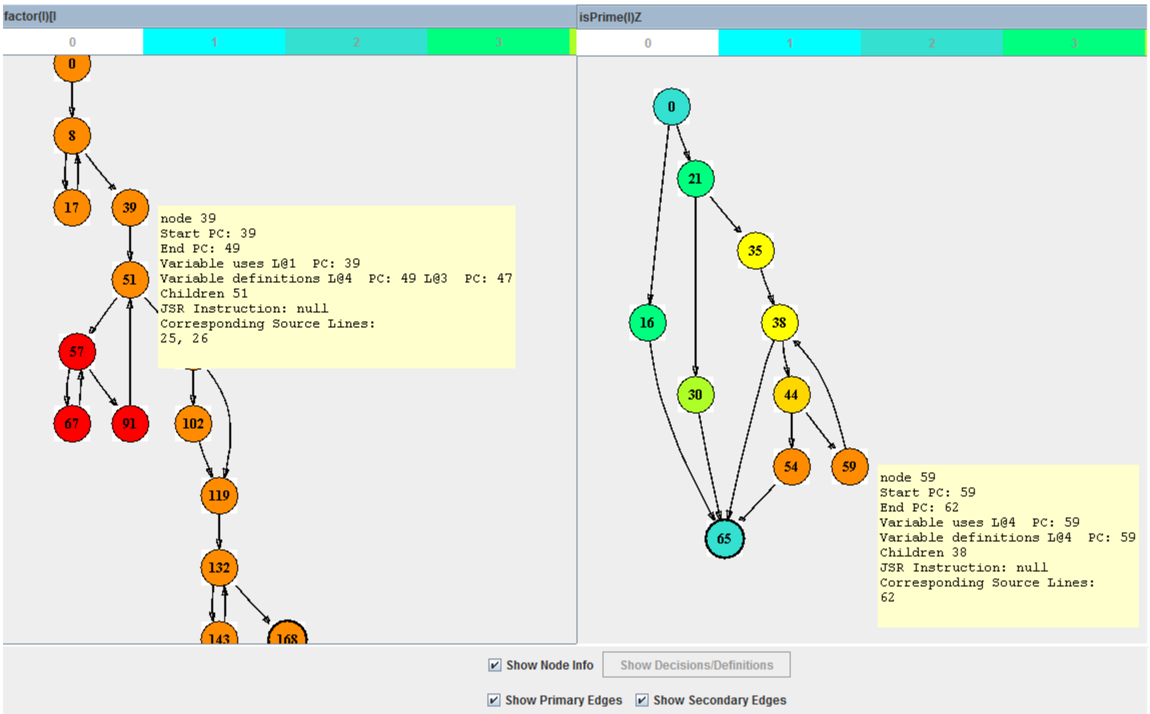
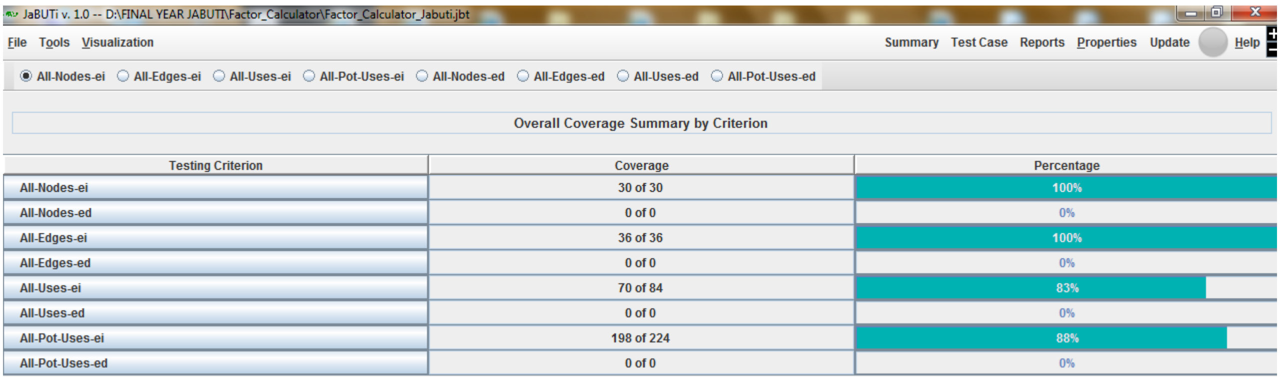
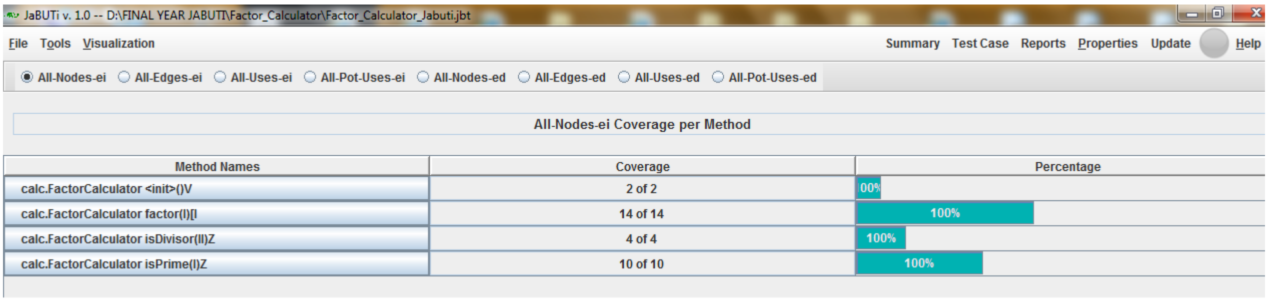


Figure 6.1: DUG of factor() and isprime() of factor class before test runs



Testing Criterion	Coverage	Percentage
All-Nodes-ei	30 of 30	100%
All-Nodes-ed	0 of 0	0%
All-Edges-ei	36 of 36	100%
All-Edges-ed	0 of 0	0%
All-Uses-ei	70 of 84	83%
All-Uses-ed	0 of 0	0%
All-Pot-Uses-ei	198 of 224	88%
All-Pot-Uses-ed	0 of 0	0%

Figure 6.2: Code coverage summary for several test runs



Method Names	Coverage	Percentage
calc.FactorCalculator <init>()V	2 of 2	00%
calc.FactorCalculator factor(I)[I	14 of 14	100%
calc.FactorCalculator isDivisor(II)Z	4 of 4	100%
calc.FactorCalculator isPrime(I)Z	10 of 10	100%

Figure 6.3: Method coverage of factor calculator for several test runs

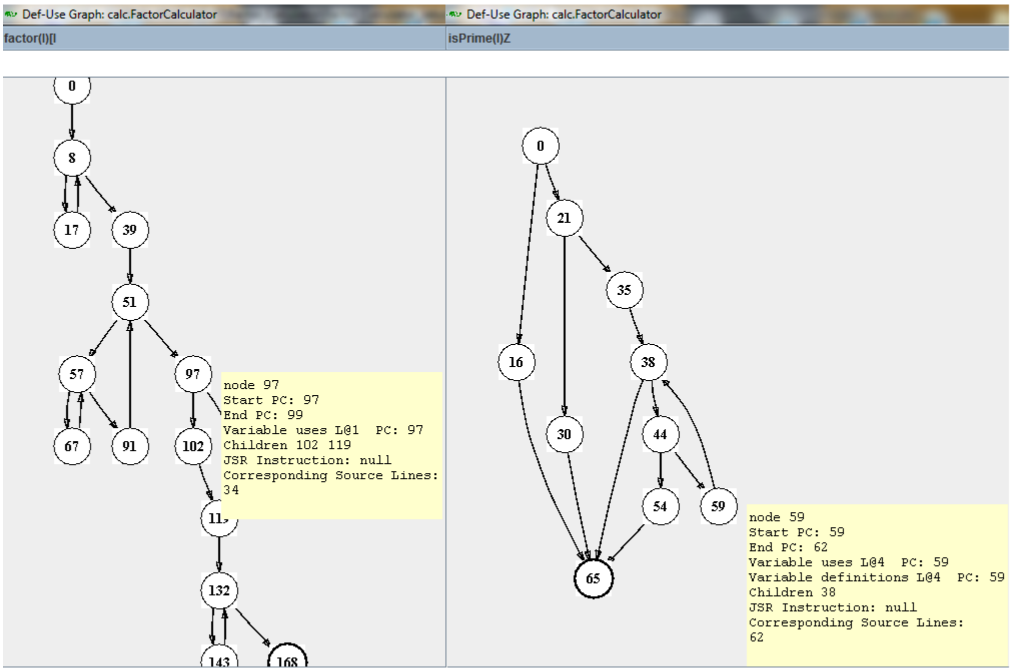


Figure 6.4: DUG of factor() and isprime() of factor class after coverage

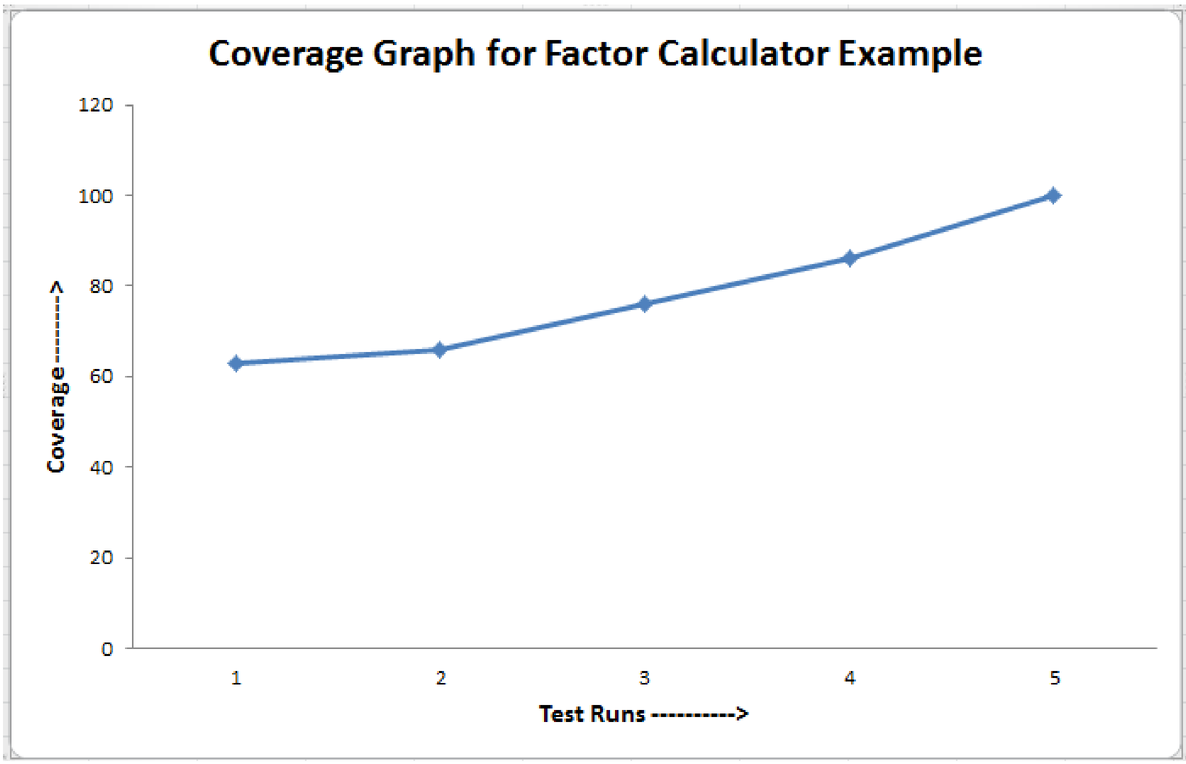


Figure 6.5: Coverage graph of Factor-calculator example

Table 6.1: Comparison between Code coverage and test runs for Factor Calculator

Test Run ID	Code Coverage (in %)
1	63
2	66
3	76
4	86
5	100

6.2.2 Vending Machine Example

This example comprises of three classes TestDriver, Dispenser and VendingMAchine. Out of these three classes Dispenser and VendingMachine classes are instrumented. Dispenser class consists of three methods dispense() which takes two parameters, available and setValidSelection() which takes an array as parameter.

Total number of Classes = 2

Total lines of source code = 122

Table 6.2: Comparison between Code coverage and test runs for Vending Machine

Test Run ID	Code Coverage (in %)
1	44
2	44
3	62
4	65
5	72
6	75
7	79
8	82
9	93
10	96

Screenshots

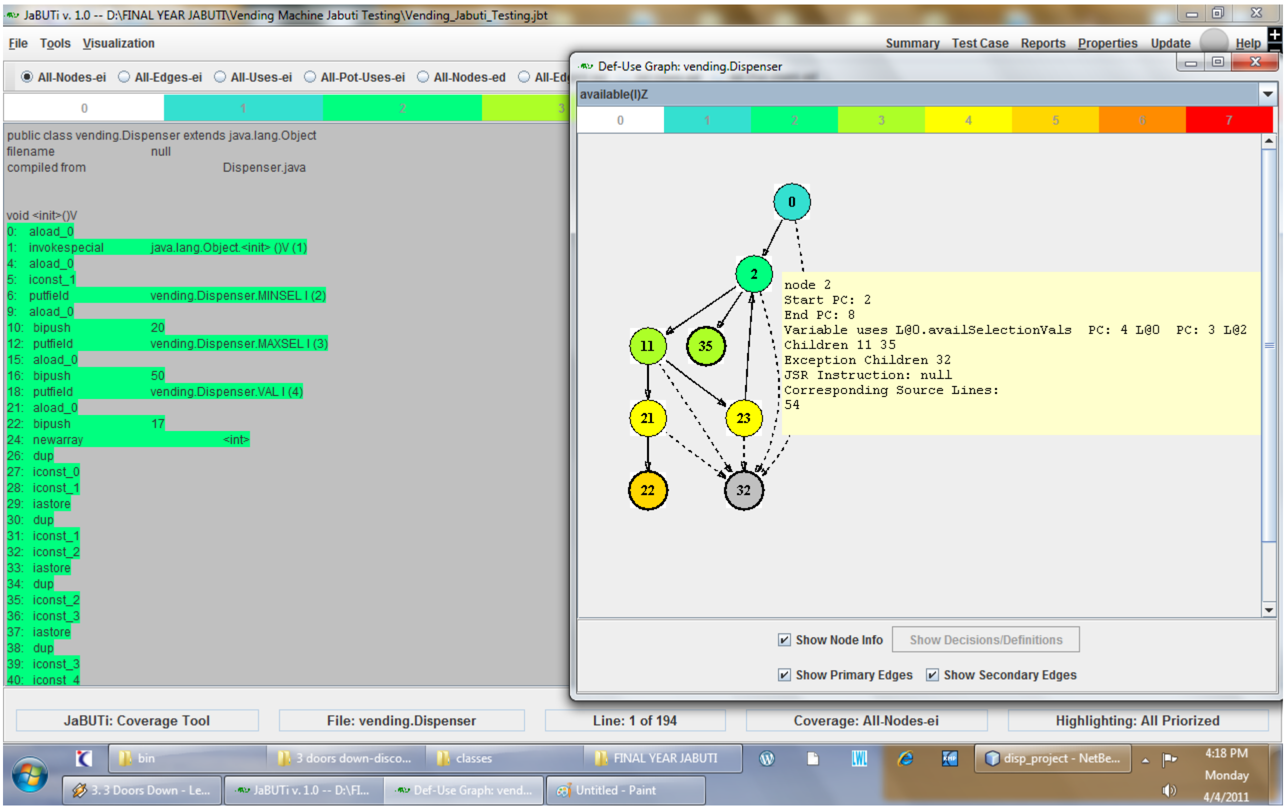


Figure 6.6: DUG of available function of dispenser class before test runs

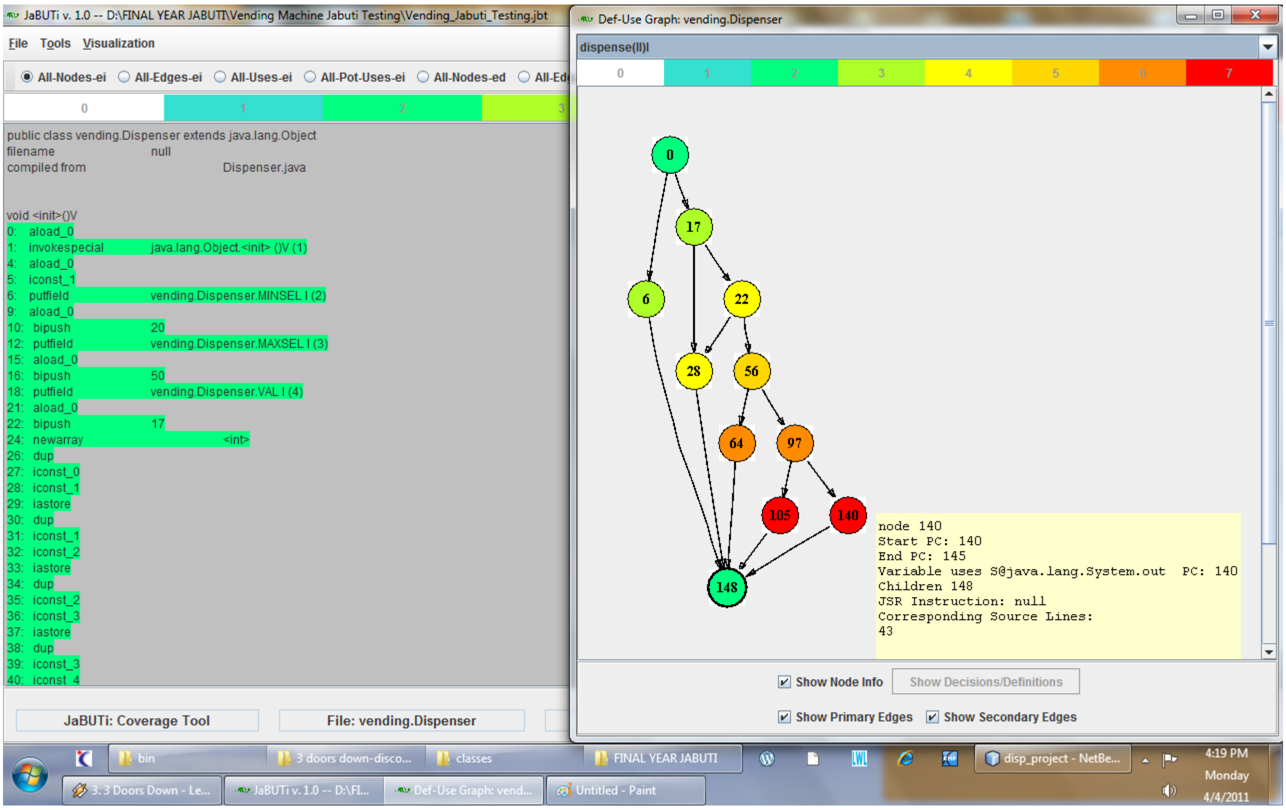


Figure 6.7: DUG of dispense function of dispenser class before test runs

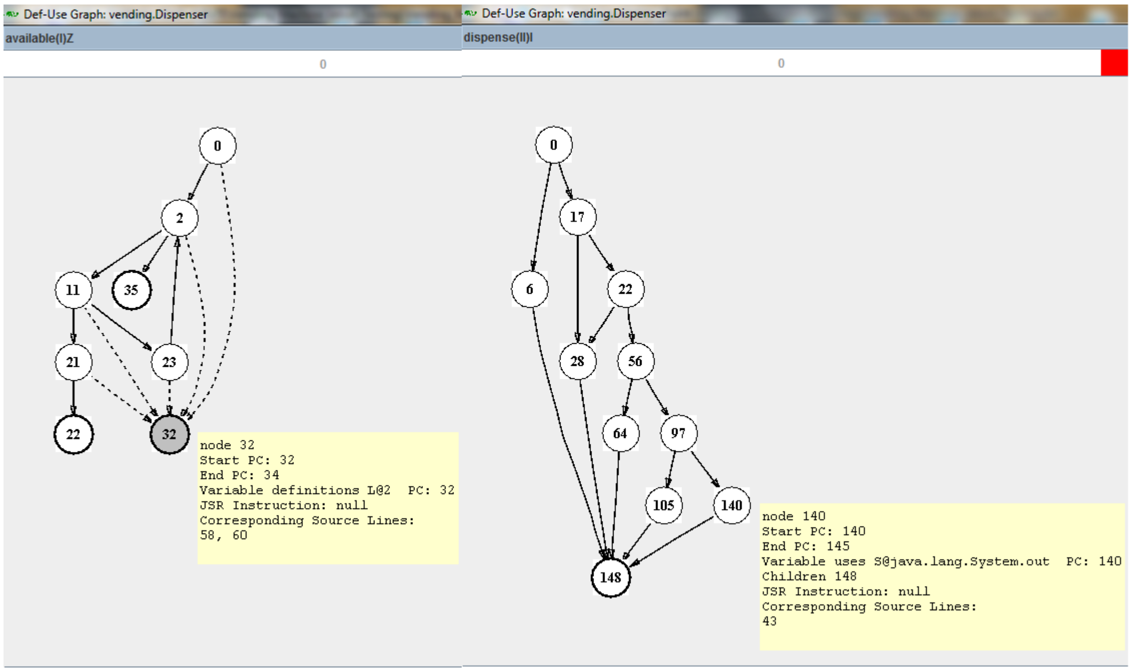


Figure 6.8: DUG of available() and dispense() functions of dispenser class after coverage

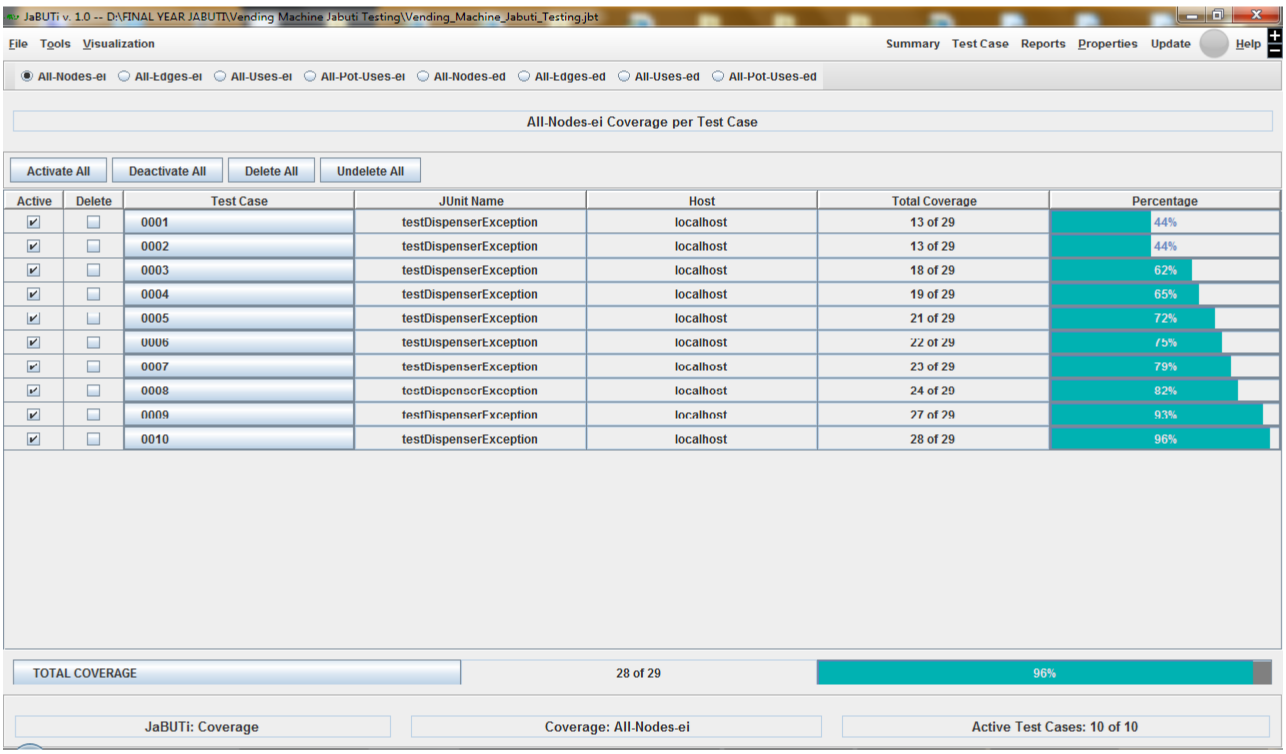
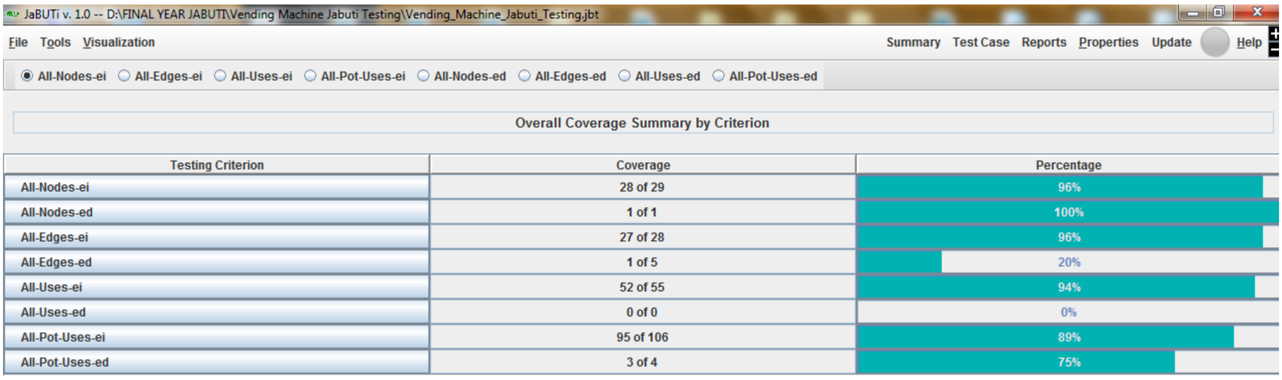


Figure 6.9: Code coverage summary of vending machine for different test runs



Testing Criterion	Coverage	Percentage
All-Nodes-ei	28 of 29	96%
All-Nodes-ed	1 of 1	100%
All-Edges-ei	27 of 28	96%
All-Edges-ed	1 of 5	20%
All-Uses-ei	52 of 55	94%
All-Uses-ed	0 of 0	0%
All-Pot-Uses-ei	95 of 106	89%
All-Pot-Uses-ed	3 of 4	75%

Figure 6.10: Code coverage summary by criteria for several test runs

Method Names	Coverage	Percentage
vending.Dispenser <init>()V	2 of 2	100%
vending.Dispenser available(I)Z	7 of 7	100%
vending.Dispenser dispense(II)I	11 of 11	[illegible]
vending.Dispenser setValidSelection([I)V	1 of 1	100%
vending.VendingMachine <init>()V	2 of 2	100%
vending.VendingMachine insertCoin()V	1 of 1	100%
vending.VendingMachine returnCoin()V	4 of 4	100%
vending.VendingMachine vendItem(I)V	0 of 1	[illegible]

Figure 6.11: Method coverage summary of vending machine for different test runs

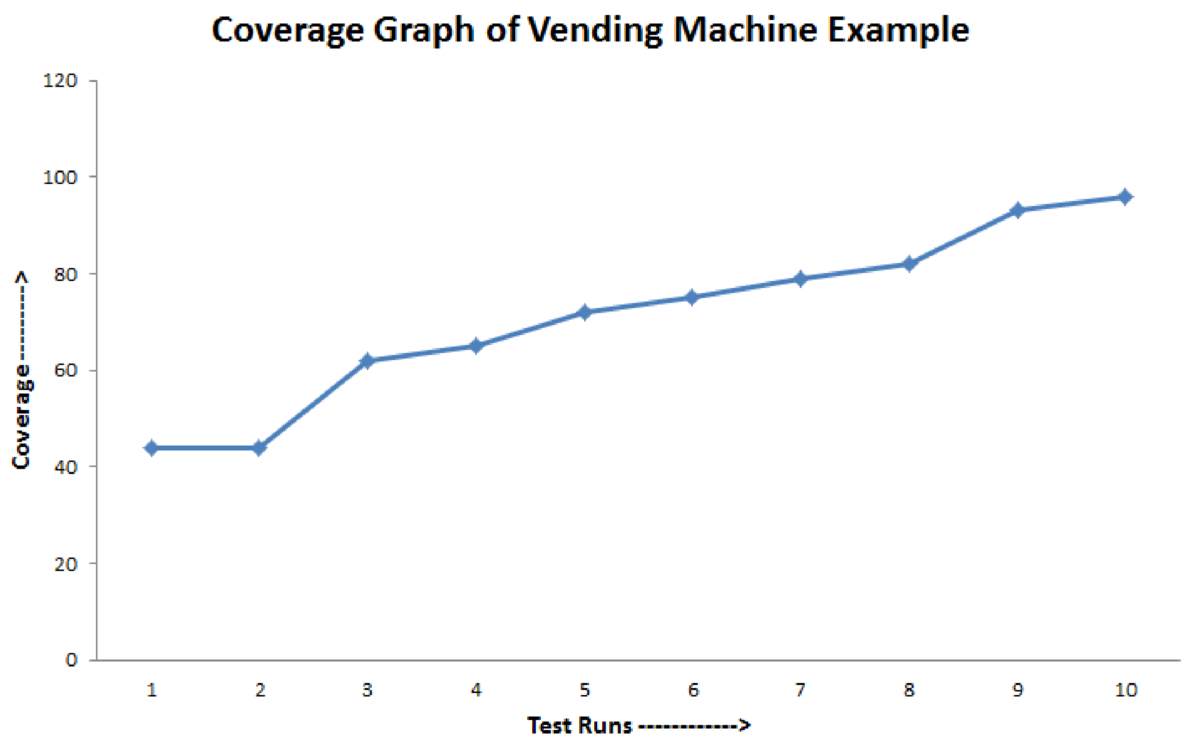


Figure 6.12: Coverage graph of Vending Machine example

6.2.3 Subject Record Example

This example comprises of two classes record.java and subject.java. record.java consists of three important functions named getStatus(), getSub(), getGrade(). getSub() returns the subject name. getGrade returns the grade obtained by the student accorting to the marks obtained in exams. getStatus returns the status, whether the student is passed or not according to the grades obtained by the student. Subject class on the other hand consists of five methods namely getTa(), getMid(), getEnd(), getSub(), getTotal(). getTa() method returns the teacher assessment marks obtained by a student. getMid() returns the mid-semester marks obtained by the student. getEnd() returns the end-semester marks obtained by the student. getTotal() returns the total marks, which is the combination of mid-semester, end-semester and teachers assessment marks. getSub() returns the subject name.

Total number of Classes = 2

Total lines of source code = 207

Table 6.3: Comparison between Code coverage and test runs for Subject-Record

Test Run ID	Code Coverage (in %)
1	7
2	10
3	14
4	17
5	21
6	24
7	26
8	33
9	35
10	36
11	42
12	45
13	61
14	64
15	68
16	71
17	75
18	78
19	82
20	84
21	96
22	98

Screenshots

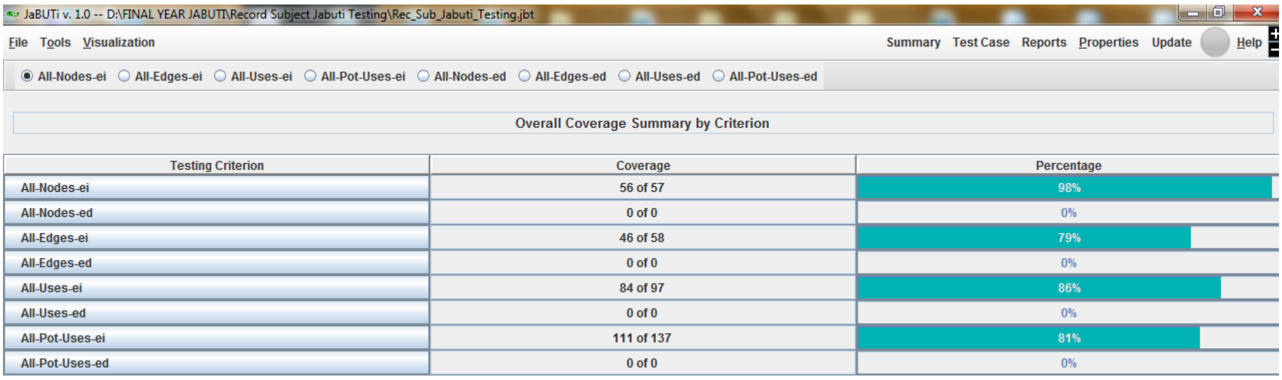


Figure 6.13: Code coverage summary by criteria for several test runs

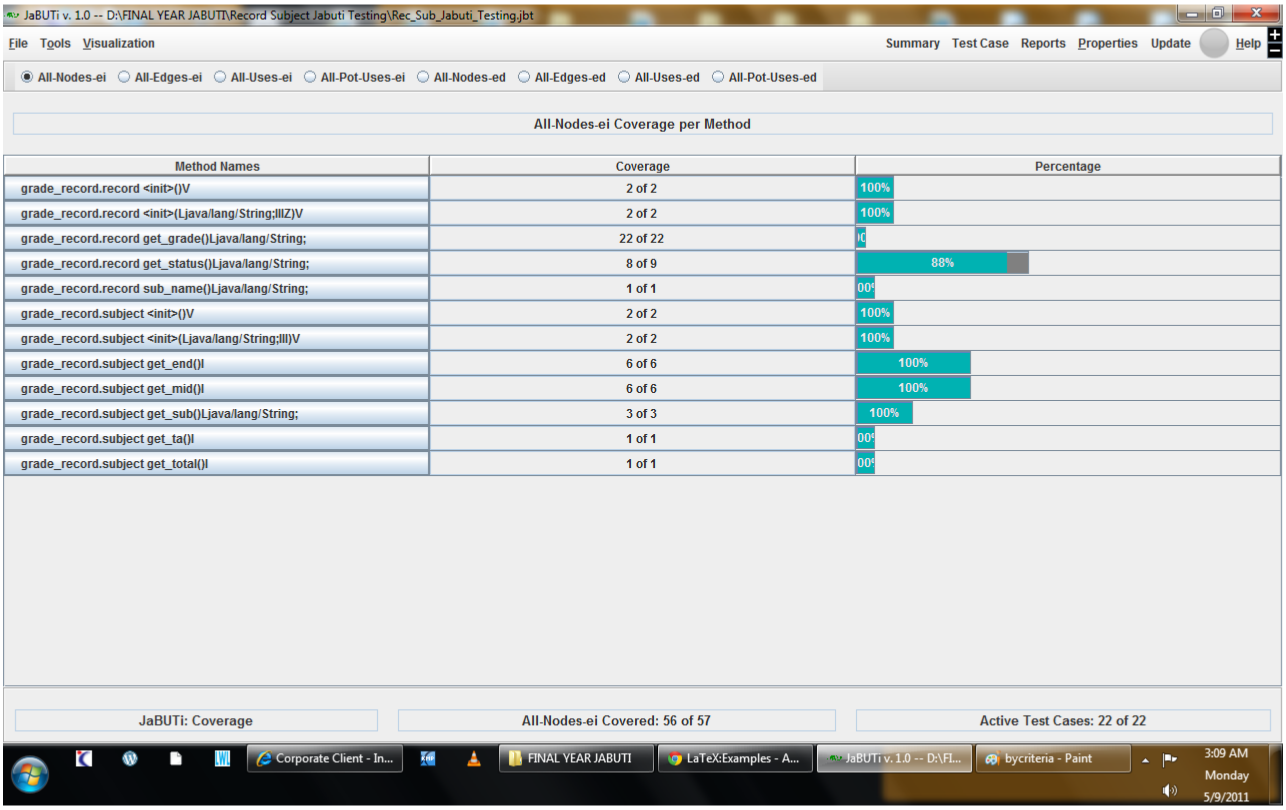


Figure 6.14: Method coverage summary of subject-record for different test runs

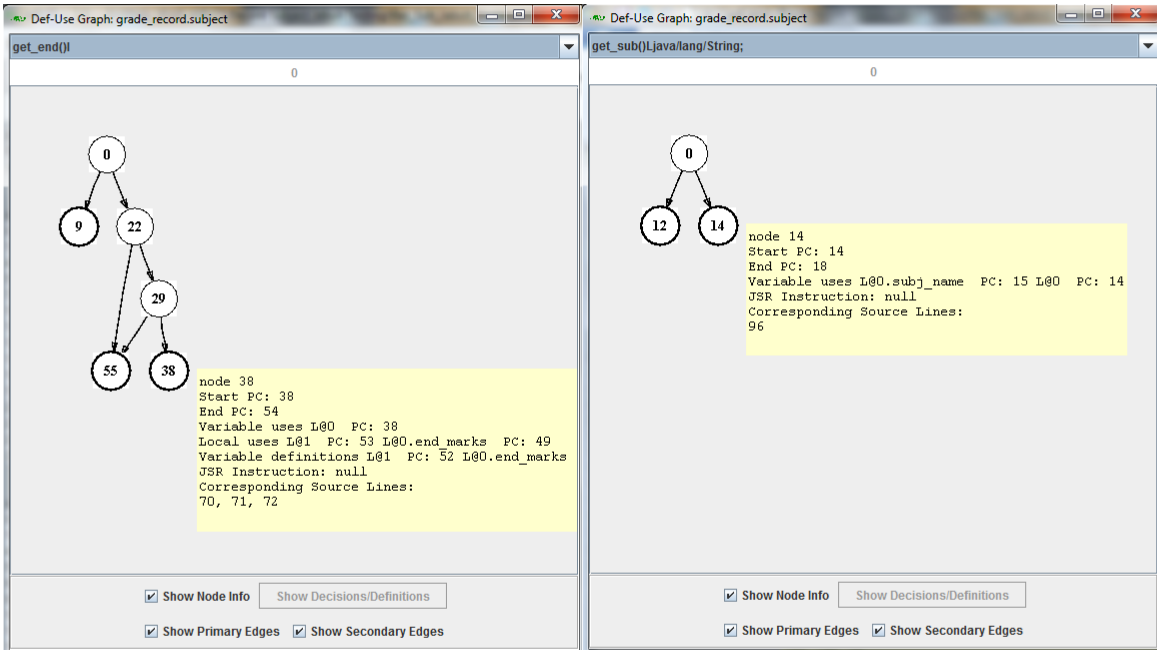


Figure 6.15: DUG of getend() and getsub() functions of subject class

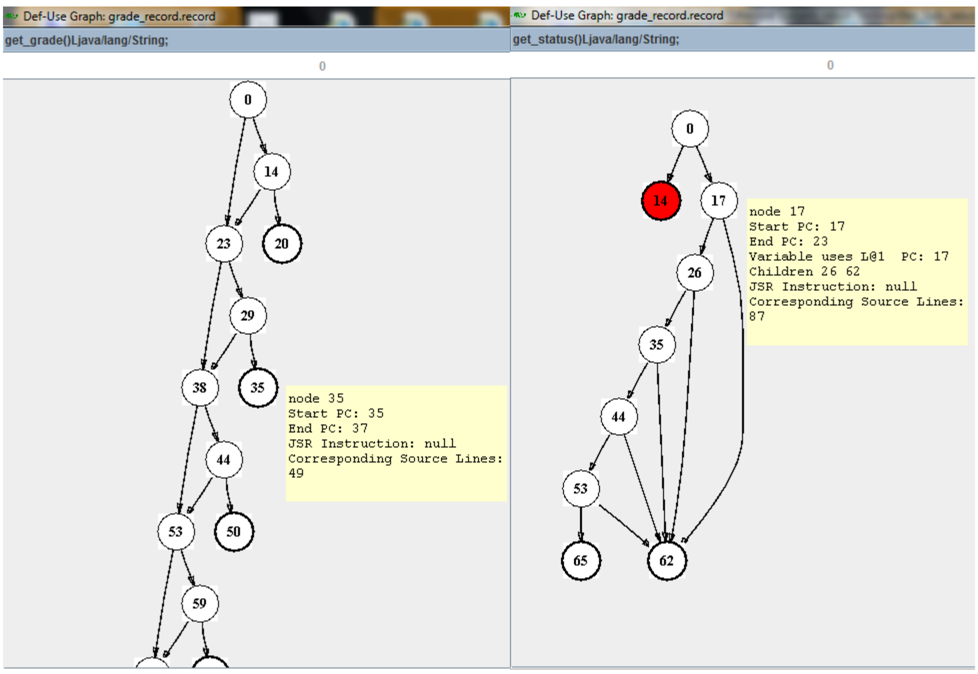
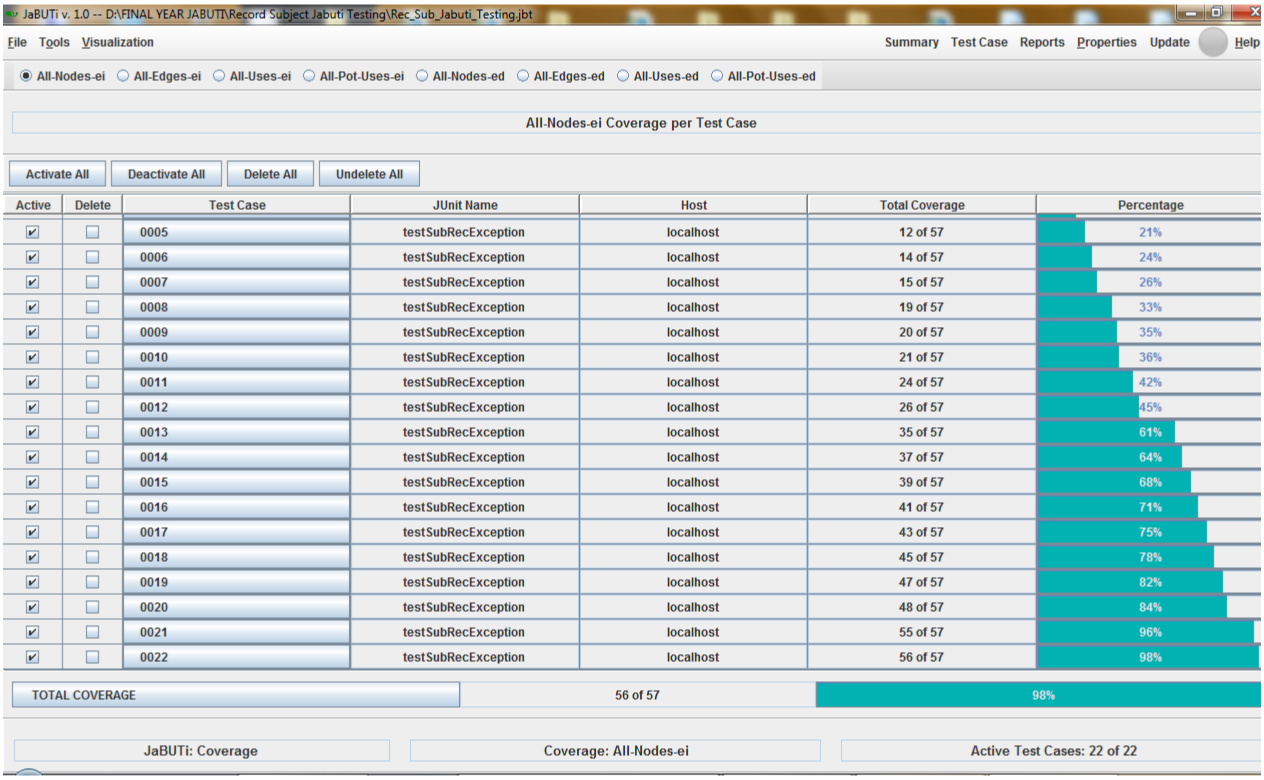


Figure 6.16: DUG of getgrade() and getstatus() functions of record class



Active	Delete	Test Case	JUnit Name	Host	Total Coverage	Percentage
☑	☐	0005	testSubRecException	localhost	12 of 57	21%
☑	☐	0006	testSubRecException	localhost	14 of 57	24%
☑	☐	0007	testSubRecException	localhost	15 of 57	26%
☑	☐	0008	testSubRecException	localhost	19 of 57	33%
☑	☐	0009	testSubRecException	localhost	20 of 57	35%
☑	☐	0010	testSubRecException	localhost	21 of 57	36%
☑	☐	0011	testSubRecException	localhost	24 of 57	42%
☑	☐	0012	testSubRecException	localhost	26 of 57	45%
☑	☐	0013	testSubRecException	localhost	35 of 57	61%
☑	☐	0014	testSubRecException	localhost	37 of 57	64%
☑	☐	0015	testSubRecException	localhost	39 of 57	68%
☑	☐	0016	testSubRecException	localhost	41 of 57	71%
☑	☐	0017	testSubRecException	localhost	43 of 57	75%
☑	☐	0018	testSubRecException	localhost	45 of 57	78%
☑	☐	0019	testSubRecException	localhost	47 of 57	82%
☑	☐	0020	testSubRecException	localhost	48 of 57	84%
☑	☐	0021	testSubRecException	localhost	55 of 57	96%
☑	☐	0022	testSubRecException	localhost	56 of 57	98%
TOTAL COVERAGE					56 of 57	98%

Figure 6.17: Code coverage summary of record-subject for different test runs

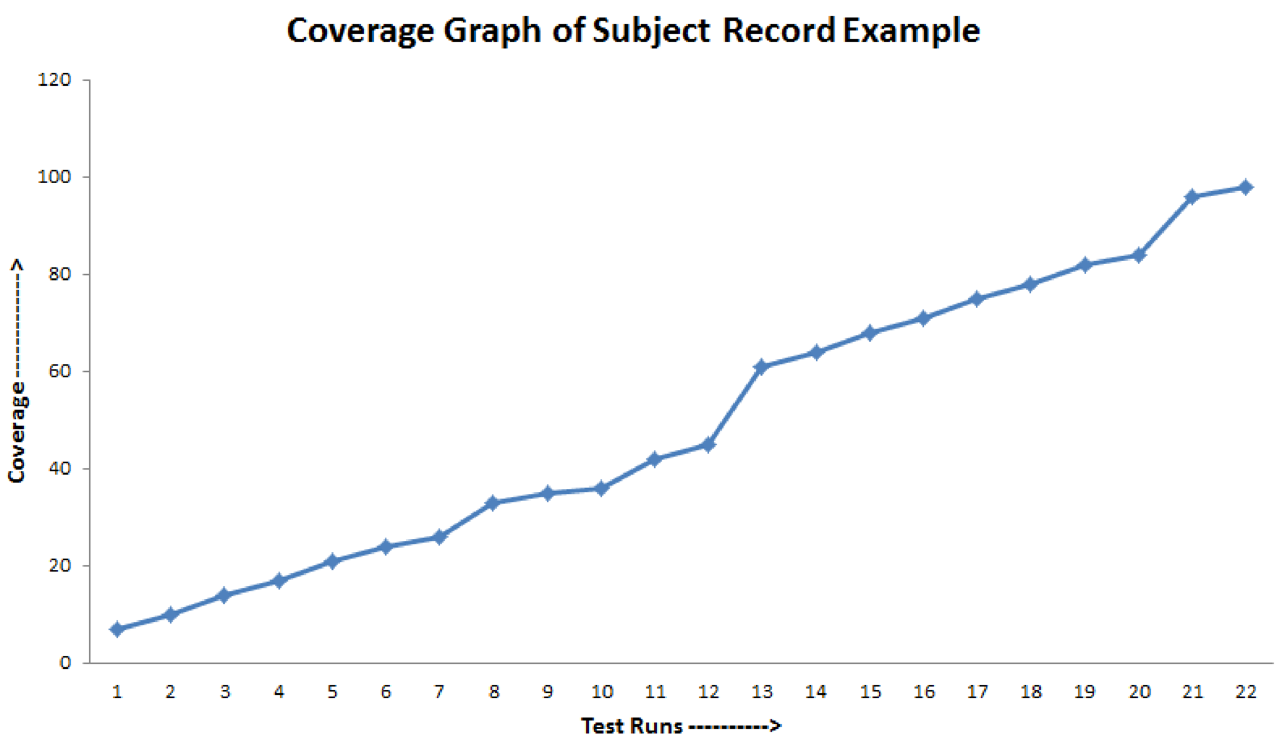


Figure 6.18: Coverage graph of Subject-Record example

6.2.4 Banking System Project Example

This example consists of five classes names bank.java, loan.java, account.java, bankbranch.java, customer.java. Each of the classes contains different functions. Bank class have some basic functions like bankAddress(), bankName(), dispBank(). Bankbranch class contains functions like setValues(), bankBranchDetails(), branchAddress(). Loan class contains methods like getLoanType(), getLoanAmount(), setLoanAmount(), setInterest(), getInterest(), dispLoanInfo(). Account class contains methods like changeType(), dispAccountInfo(), getTotalBal(). Customer class contains methods like getBalance(), depositAmount(), withdrawAmount(), borrowLoan(), totalLoanAmountReturned(), dispCustomerDetails().

Total number of Classes = 5

Total lines of source code = 345

Screenshots

Table 6.4: Comparison between Code coverage and test runs for Banking System

Test Run ID	Code Coverage (in %)
1	4
2	6
3	8
4	12
5	14
6	29
7	30
8	32
9	33
10	38
11	45
12	46
13	48
14	50
15	51
16	56
17	58
18	61
19	62
20	67
21	70
22	77
23	79
24	80
25	91

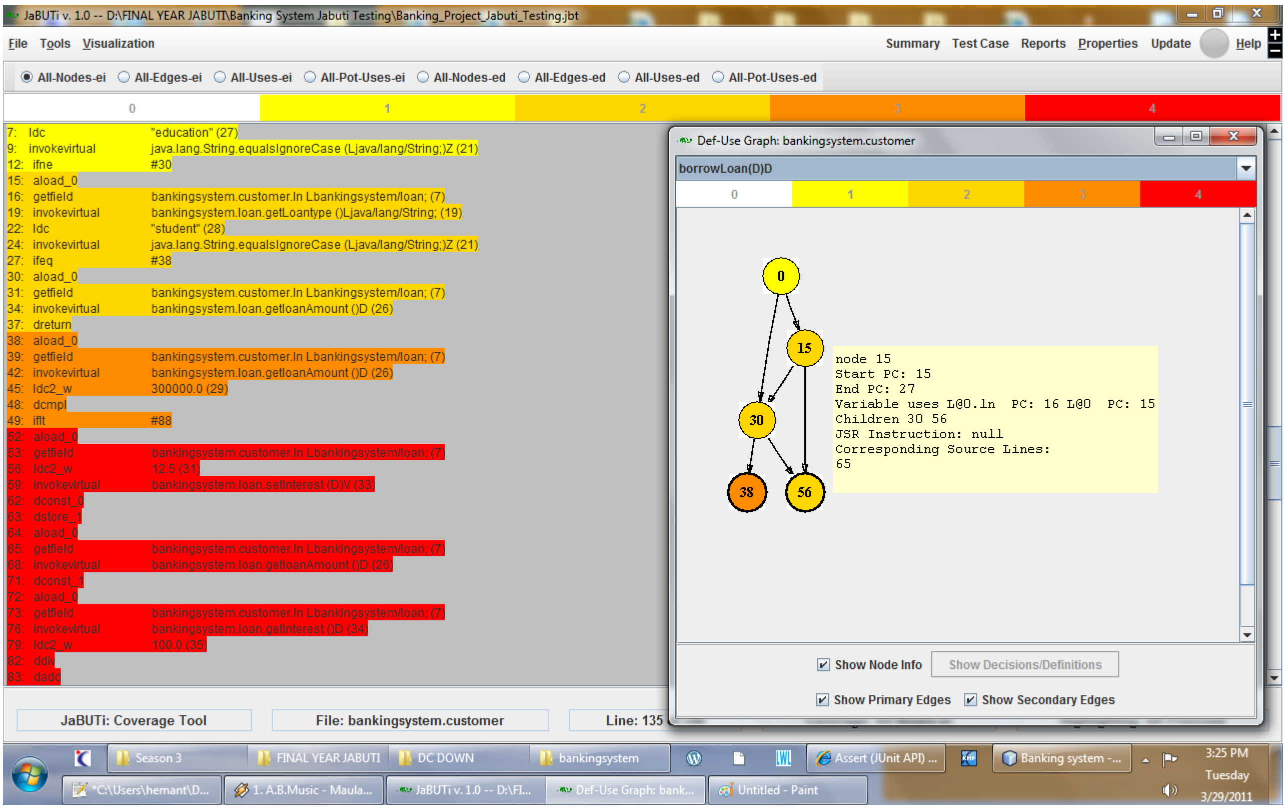


Figure 6.19: DUG of borrowLoan()function of customer class before test runs

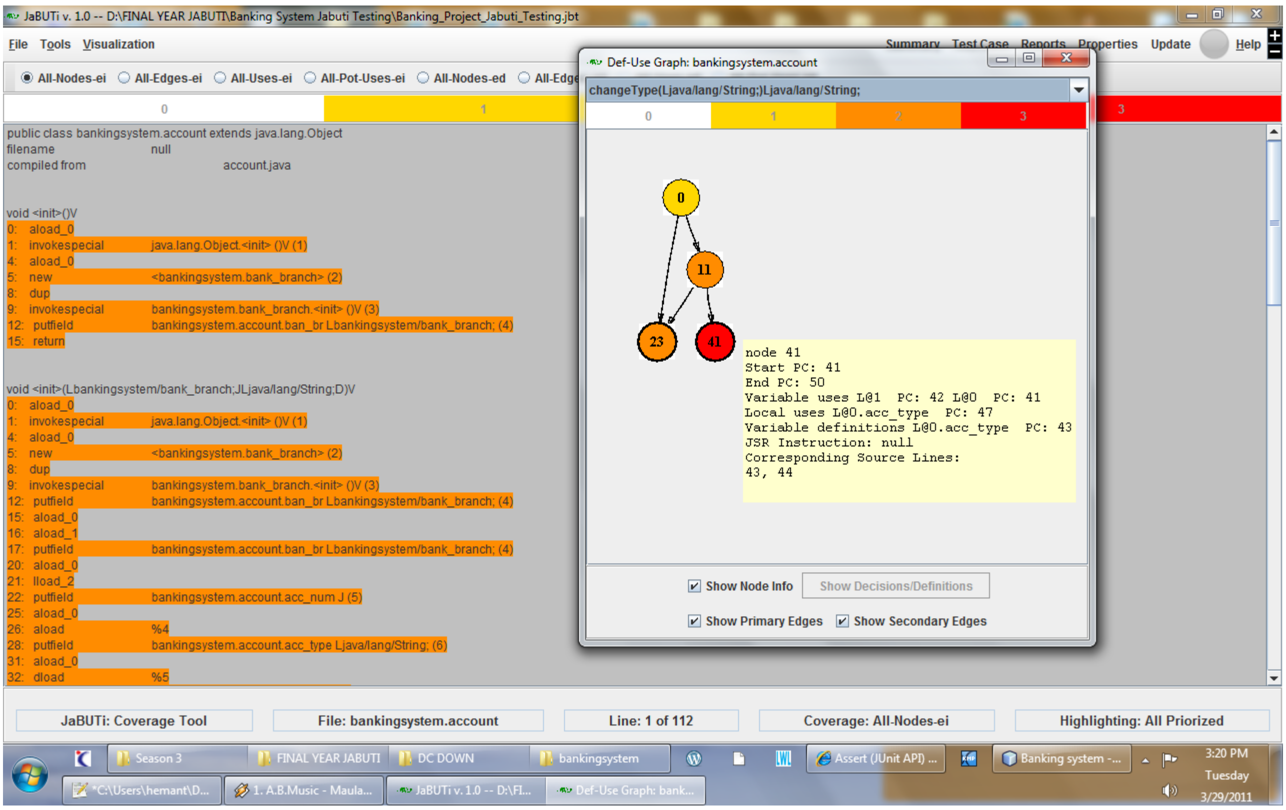


Figure 6.20: DUG of changeType()function of account class before test runs

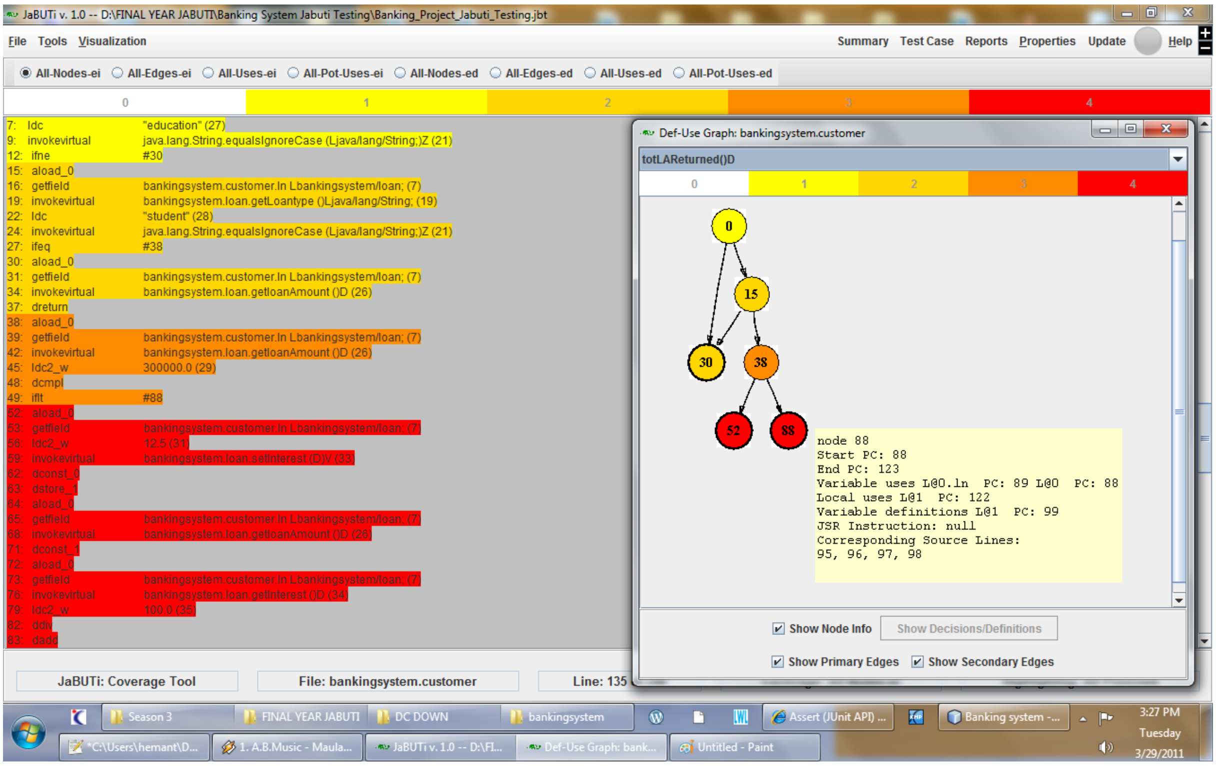


Figure 6.21: DUG of totalLoan()function of customer class before test runs

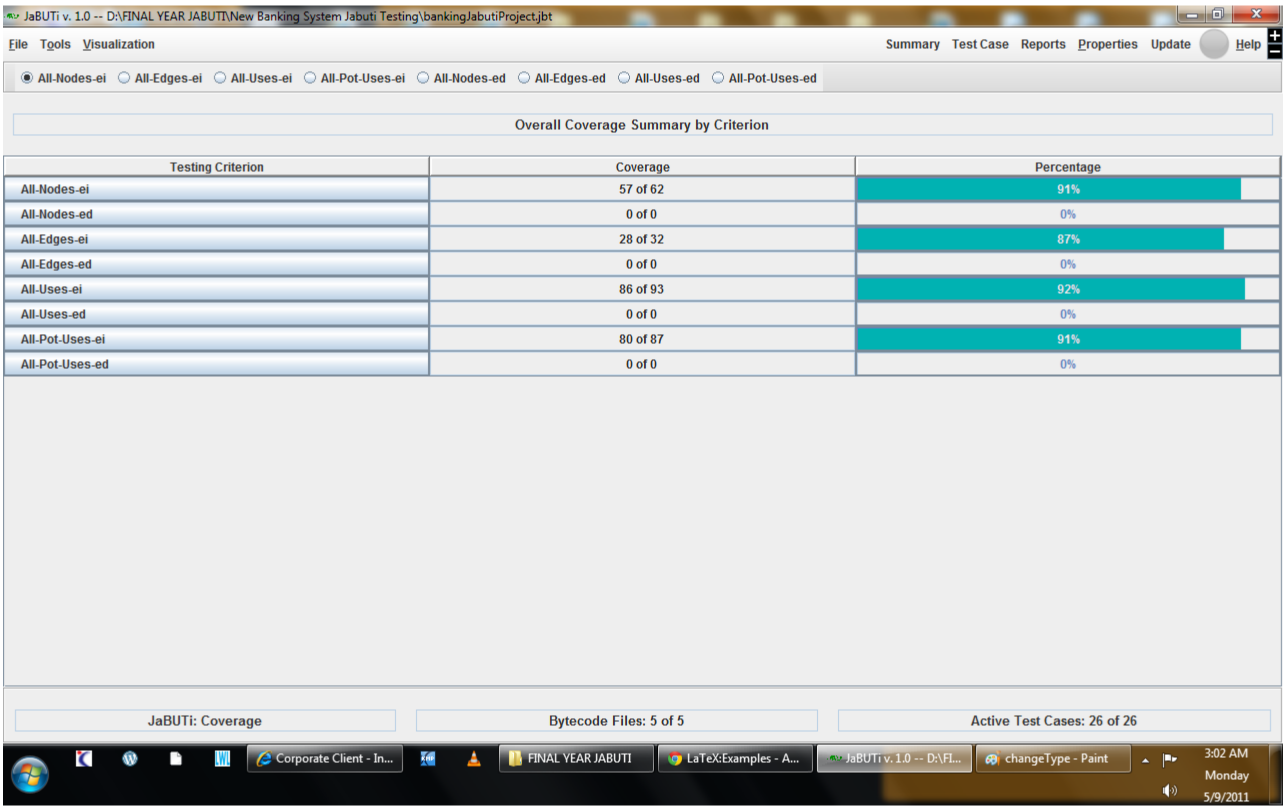


Figure 6.22: Code coverage summary by criteria for several test runs

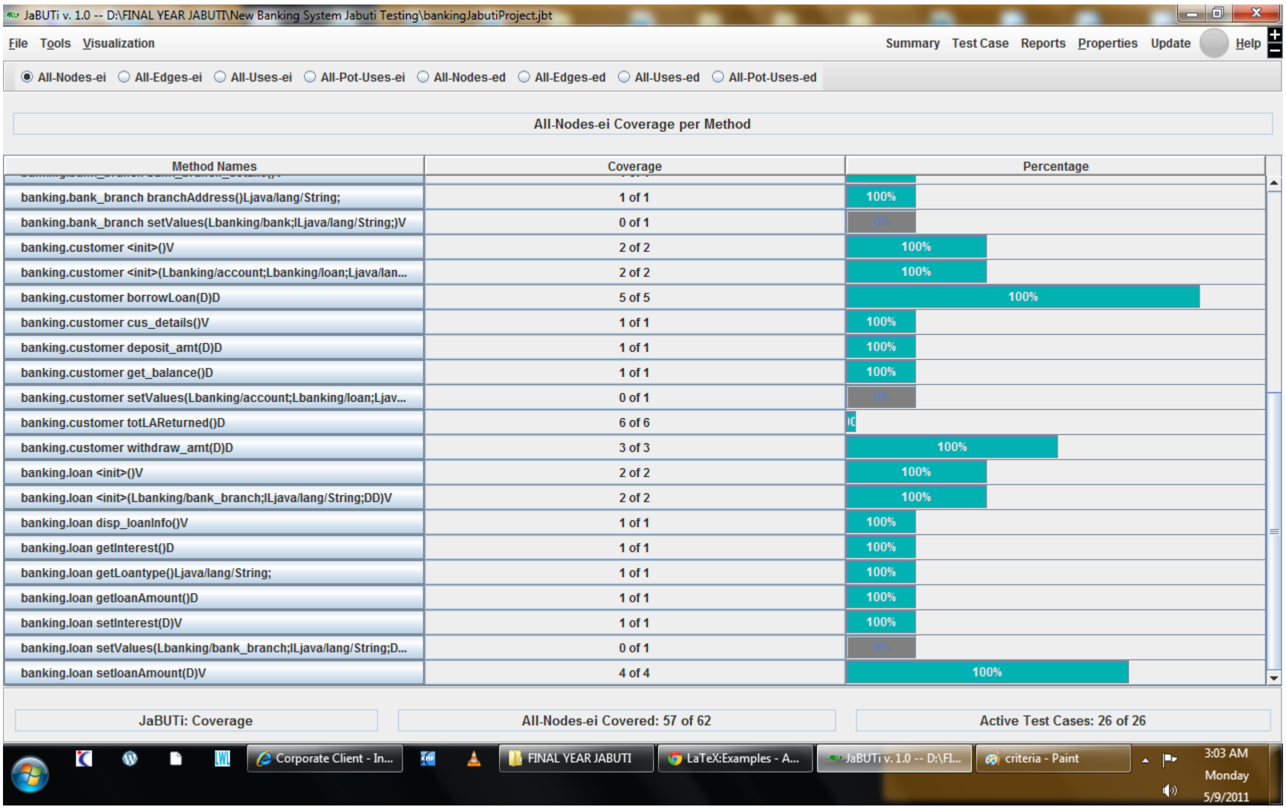


Figure 6.23: Method coverage summary of Banking system for different test runs

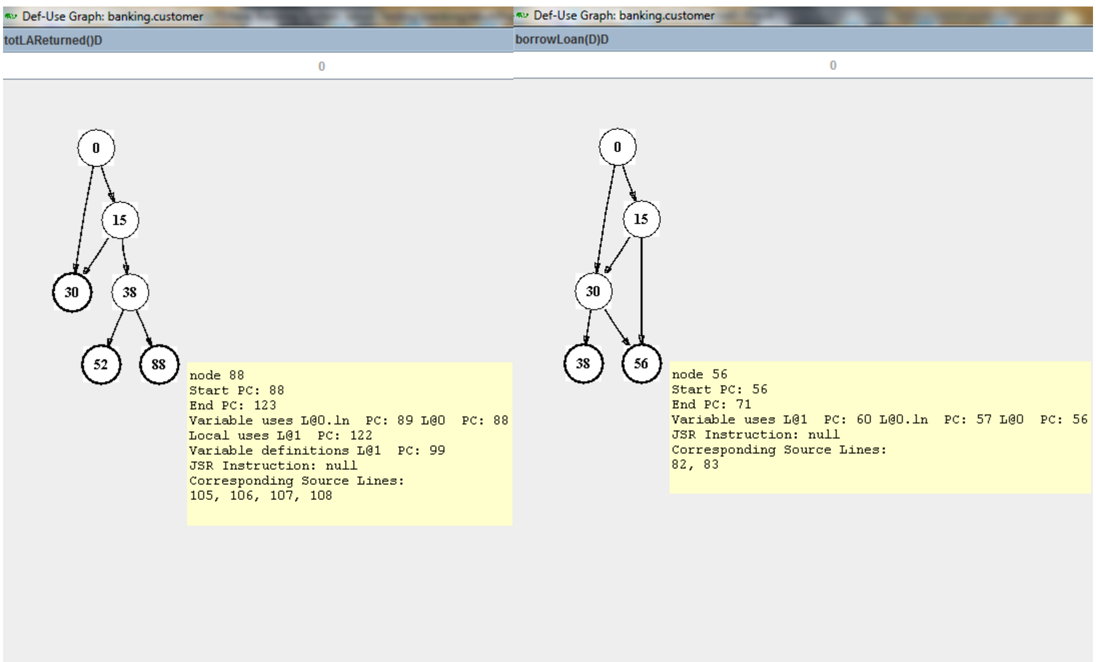


Figure 6.24: DUG of totalloanreturned() and borrowloan() functions of customer class after coverage

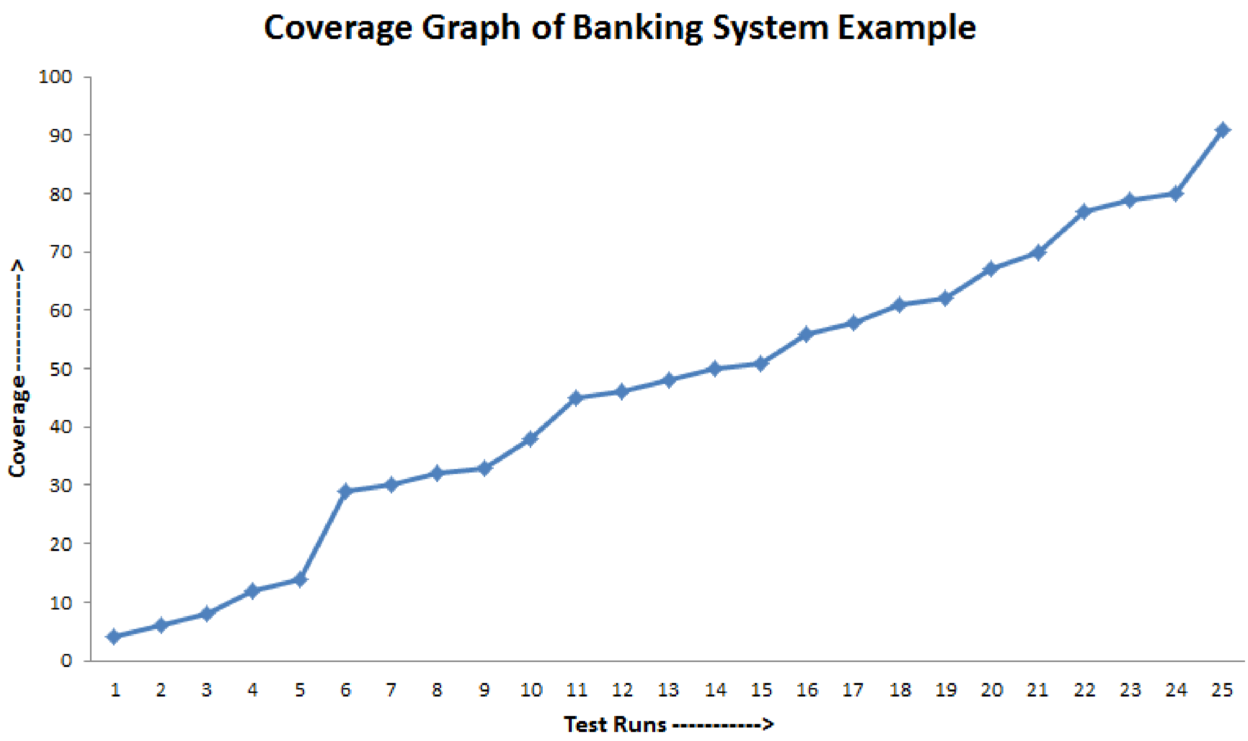


Figure 6.25: Coverage graph of banking system example

6.2.5 Overall Assessment

Overall result of different cases studies are shown here

Table 6.5: Comparison between Code coverage and test runs for Overall Assessment

Example	No. of Test Runs	Coverage (in %)
Factor Calculator	5	100
Vending Machine	10	96
Subject Record	22	98
Banking System	26	91

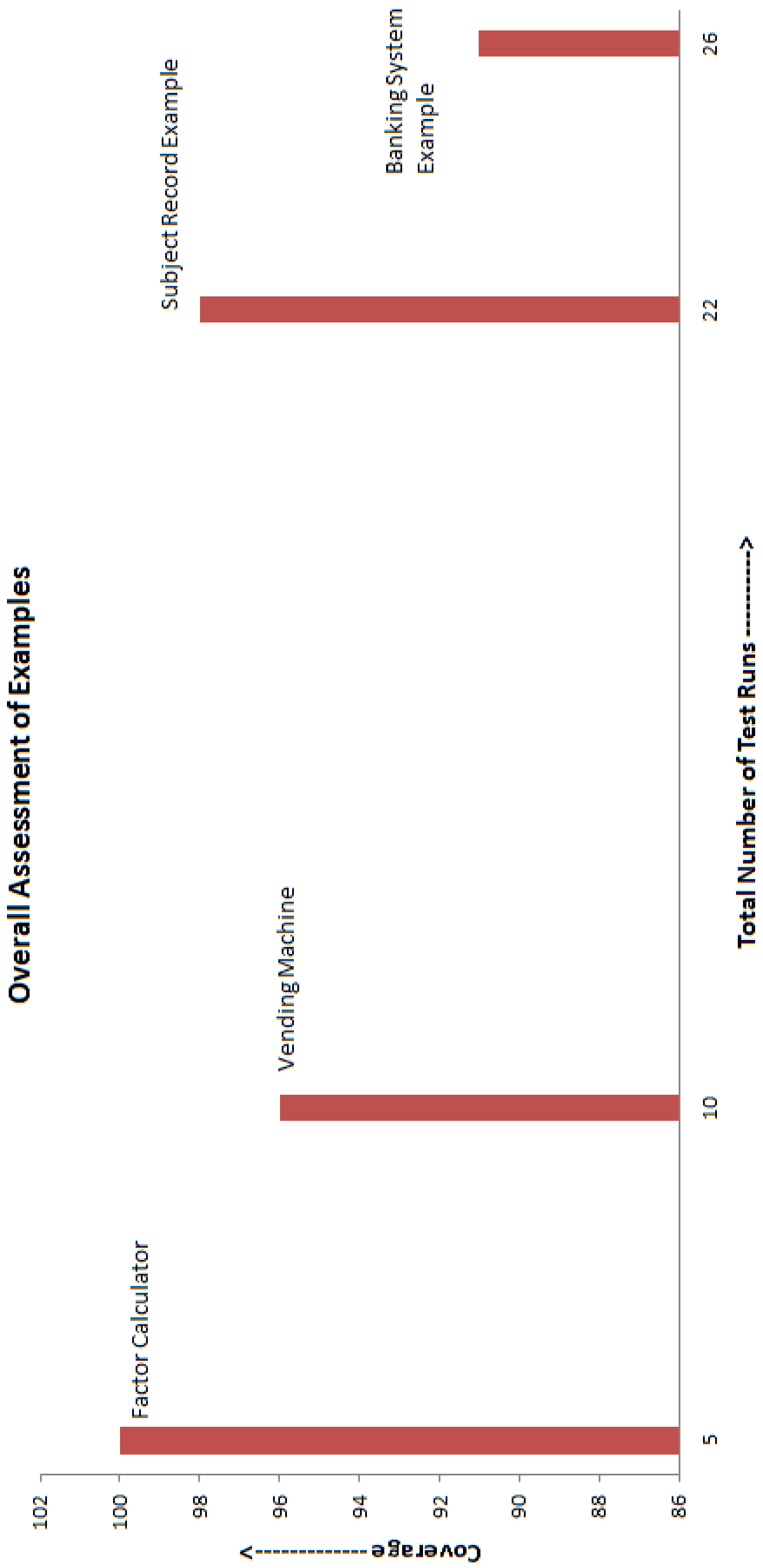


Figure 6.26: Coverage graph of all cases for Overall assessment

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have proposed an Improved Code Coverage Algorithm to obtain high code coverage of Java programs in minimum test runs. We have implemented our algorithm for the code coverage analysis of various case studies. We have used the All-Nodes-ei criteria for coverage in all the case studies. Coverage of each example has been exercised until it exceeds 90 %. Code coverage for each case study is summarised by various criteria such as by method, by class etc. At last comparision is made between the code covered and various by other criteria like by class, by methods, by other criteria.

7.2 Future Work

We have done the code coverage analysis mostly on All-Nodes-ei criteria. Our proposed algorithm can further be implemented for the code coverage analysis of Java programs based on other criteria like All-edges-ei, All-uses-ei, All-Pot-Uses-ei, All-Nodes-ed, All-edges-ed, All-uses-ed, All-Pot-Uses-ed. Code coverage based on these criteria can further be increased to required level %. Our Algorithm for Code Coverage estimation, using JaBUTi, can further be implemented on programs containing GUI components. It can also be enhanced in the field of Distributed system and aspect-oriented Java programming. Further, the work of JaBUTi can be extended as a slicing tool and static matrics tool.

Bibliography

[1] W. E. Perry, Effective methods for software testing, Wiley Computer Publishing (2000).

[2] B. Marick, The Craft of Software Testing, Prentice Hall Publication (1995).

[3] A.M.R. Vincenzi, J. L Maldonado, W. E Wong, M. E Delamaro, Coverage testing of Java Programs and Components, Science of Computer Programming (2005) 31-35.

[4] http://www.bullseye.com/coverage.html,Code Coverage Analysis.

[5] Wolfgang Pree,Component-Based Software Development- A New Paradigm,MIT Press (1997) 169- 174.

[6] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, in: Proceedings of the IEEE Transactions on Software Engineering 11 (4) (1985) 367 375.

[7] M.J. Harrold, G. Rothermel, Performing data flow testing on classes, in: Proceedings of Second ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press (1994) 154- 163.

[8] G.J. Myers, The Art of Software Testing, Wiley, New York (1979).

[9] J. Zhao, Dependence analysis of Java bytecode, in: Proceedings of 24th IEEE Annual International Computer Software and Applications Conference, COMPSAC2000, IEEE Press, Taipei, Taiwan (2000) 486 491.

[10] P. Piwowarski, M. Ohba, J. Caruso, Coverage measurement experience during function test, in: Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD (1993) 287 301.

[11] B. Beizer, Black-Box Testing: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, New York (1995).

[12] J. M. Bieman, J. L. Schultz, Estimating the Number of Test Cases Required to Satisfy the All-du-paths, TAV3-SIGSOFT '89 (1989) 179- 186.

[13] Bor-Yuan Tsai, Simon Stobart, Norman Parrington, and Ian Mitchell, A State-Based Testing Approach Providing Data Flow Coverage in Object-Oriented Class Testing, in: Proceedings of The 12th International Software Quality Week Conference 1999 (QW 99), San Jose, USA (1999).

[14] Y.K. Malaiya, Michael Naixin Li, J.M. Bieman,Rick Karcich, Software Reliability Growth with Test Coverage, in: Proceedings of IEEE Transactions on Reliability, (51)(4) (2002) 420- 427.